Parallel Computing for Science & Engineering

02/23/23 & …

Instructors:
Lars Koesterke, TACC
Victor Eijkhout, TACC

Scientific Computing Terminology

Terms	Definition
NUMA	Non Uniform Memory Access. In SMP systems with multiple CPUs access time to different parts of memory may vary.
Affinity	Propensity to maintain a process or thread on a hardware execution unit.
SMP	Symmetric Multi-Process(ing/or). Single OS system with shared memory.
OpenMP – Directive	Comment statement (F90) or #pragma (C/C++) that specifies parallel operations and control.
OpenMP – Construct	The lexical extent that a directive controls.
OpenMP – Region	All code controlled by a directive– lexical extent + content of called routines.
Runtime	Code or a library within an executable that interacts with the operating system and can control code execution.

OpenMP-- Overview

- Standard is ~25 years old. Mature language
- The “language” is easily comprehended.
  You can start simple and expand.
- Light Weight from System Perspective
- Very portable –GNU and vendor compilers.
- Spend time finding parallelism can be the most difficult part. The parallelism may be hidden.
- Writing Parallel OpenMP code examples is relatively easy.
- Developing parallel algorithms and/or parallelizing serial code is much harder.
- Expert level requires awareness of scoping and synchronization.
- Expansion into other performance relevant areas like: thread pinning and memory pinning

OpenMP --- Shared Memory

- Shared Memory systems:
  - One Operating System
  - Instantiation of ONE process
  - Threads are forked (created) from within your program.
  - Multiple threads on multiple cores

What is OpenMP (Open Multi-Processing)

- **De facto standard for Scientific Parallel Programming on Symmetric Multi-Processor (SMP) Systems.**

- **It is an API (Application Program Interface) for designing and executing parallel Fortran, C and C++ programs**
    - Based on threads, but
    - Higher-level than POSIX threads (Pthreads)
      (http://www.llnl.gov/computing/tutorials/pthreads/#Abstract)
- **Implemented by:**
    - **Pragmas/comments in code**
    - **Runtime Library (interface to OS and Program Environment)**
    - **Environment Variables**
- **Compiler option required to interpret/activate directives.**

- **http://www.openmp.org/ has tutorials and description.**
- **Directed by OpenMP ARB (Architecture Review Board)**

OpenMP History

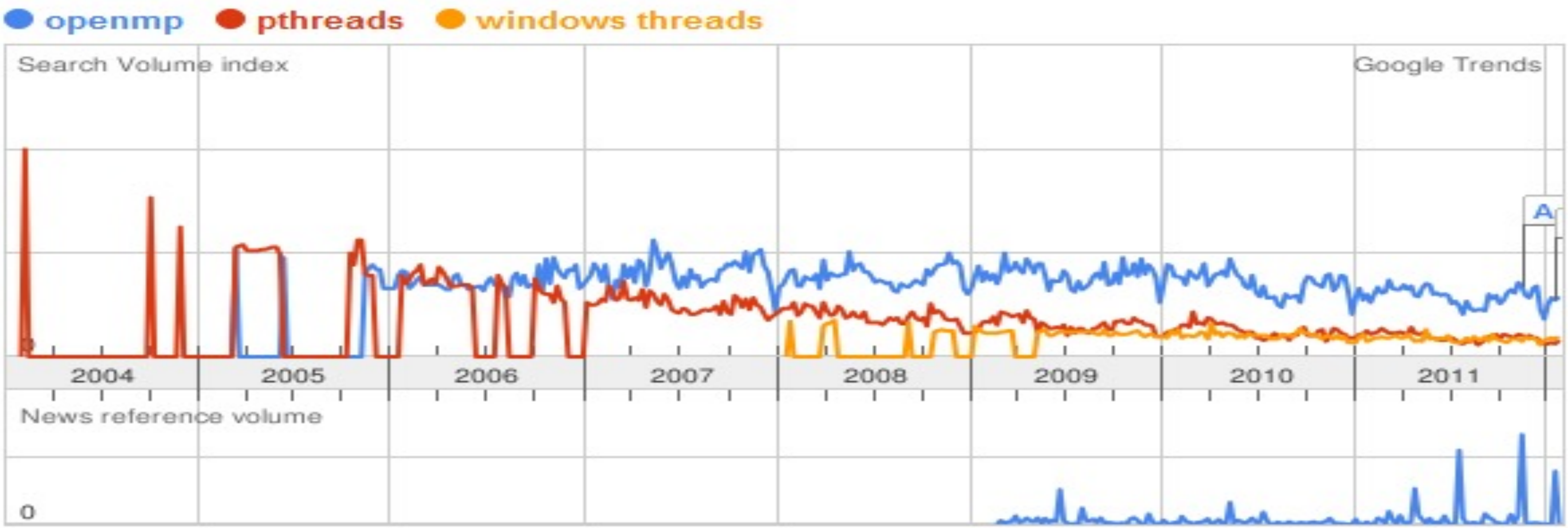
- Primary OpenMP participants

  AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, MS, TI, CAPS, NVIDIA

  ANL, LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH, TACC

- OpenMP Fortran API, Version 1.0, 1997
- OpenMP C API, Version 1.0, 1998
- OpenMP 2.0 API for Fortran, 2000
- OpenMP 2.0 API for C/C++, 2002
- OpenMP 2.5 API for C/C++ & F90 2005
- OpenMP 3.0 Tasks May 2008
- OpenMP 3.1 July 2011
- OpenMP 4.0 **Affinity, Devices, Depend, SIMD** July 2013
- OpenMP 4.5
- OpenMP 5.0

OpenMP History



OpenMP 3.0: The World is still flat, no support for NUMA (yet)!
OpenMP is hardware agnostic, it has no notion of data locality.
The Affinity problem: How to maintain or improve the nearness of threads and their most frequently used data.
Or:
Where to run threads?
Where to place data?

http://terboven.wordpress.com/

Thread binding was added in OpenMP 4.0

Advantages/Disadvantages of OpenMP

- Pros
  - Shared Memory Parallelism is easier to learn.
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
  - Converting serial code to OpenMP parallel can be easier than converting to MPI parallel.
  - Shared-Memory hardware is prevalent now.
    - Supercomputers **and** your desktop/laptop (and your phones)
    - GPUs (Graphics Cards), Multi-core CPUs
  - Complements MPI and enables full core utilization
- Cons
  - Scalability limited by memory architecture.
  - Available on Shared-Memory systems “only”.
  - Beware: “Upgrading” large serial code may be hard.

OpenMP Parallel Directives

Supports parallelism by Directives in Fortran, C/C++,…

Unlike others that require base language changes and constructs

Unlike MPI which supports parallelism through communication lib.

OpenMP implementation through the compiler

Compiler optimizes OpenMP code

TACC

Processes on a shared-memory System

- The OS starts a process
  - One instance of your computer program, the “a.out”
- Many processes may be executed on a single core through “time sharing” (time slicing).
  - The OS allows each process to run for awhile.
- The OS may run multiple processes concurrently on different cores.
- Security considerations
  - Independent processes have no direct communication (exchange of data) and are not able to read another process’s memory.
- Speed considerations
  - Time sharing among processes has a large overhead.

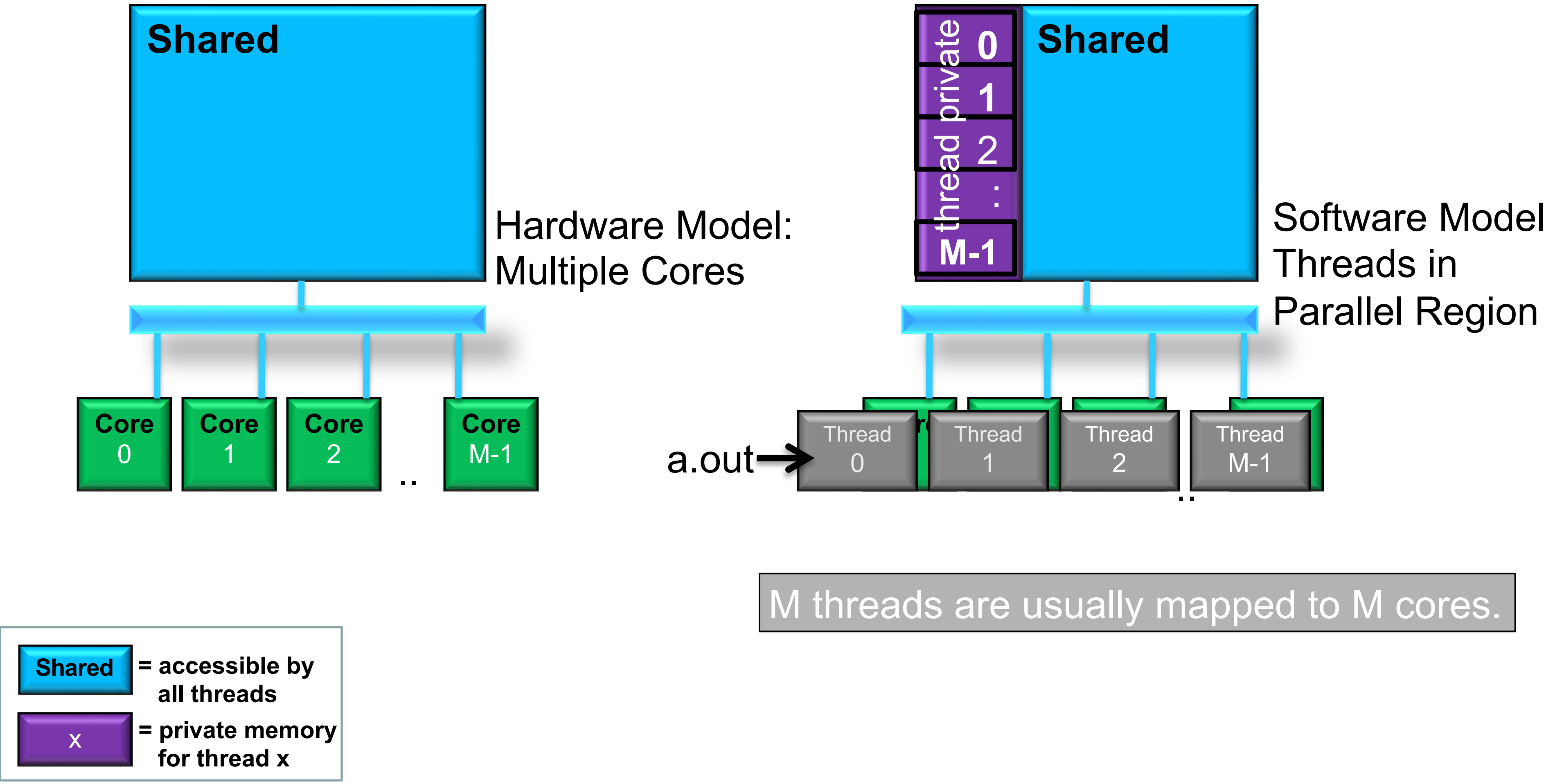
OpenMP Threads

- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process.
- Each thread is given some private memory only seen by the thread.

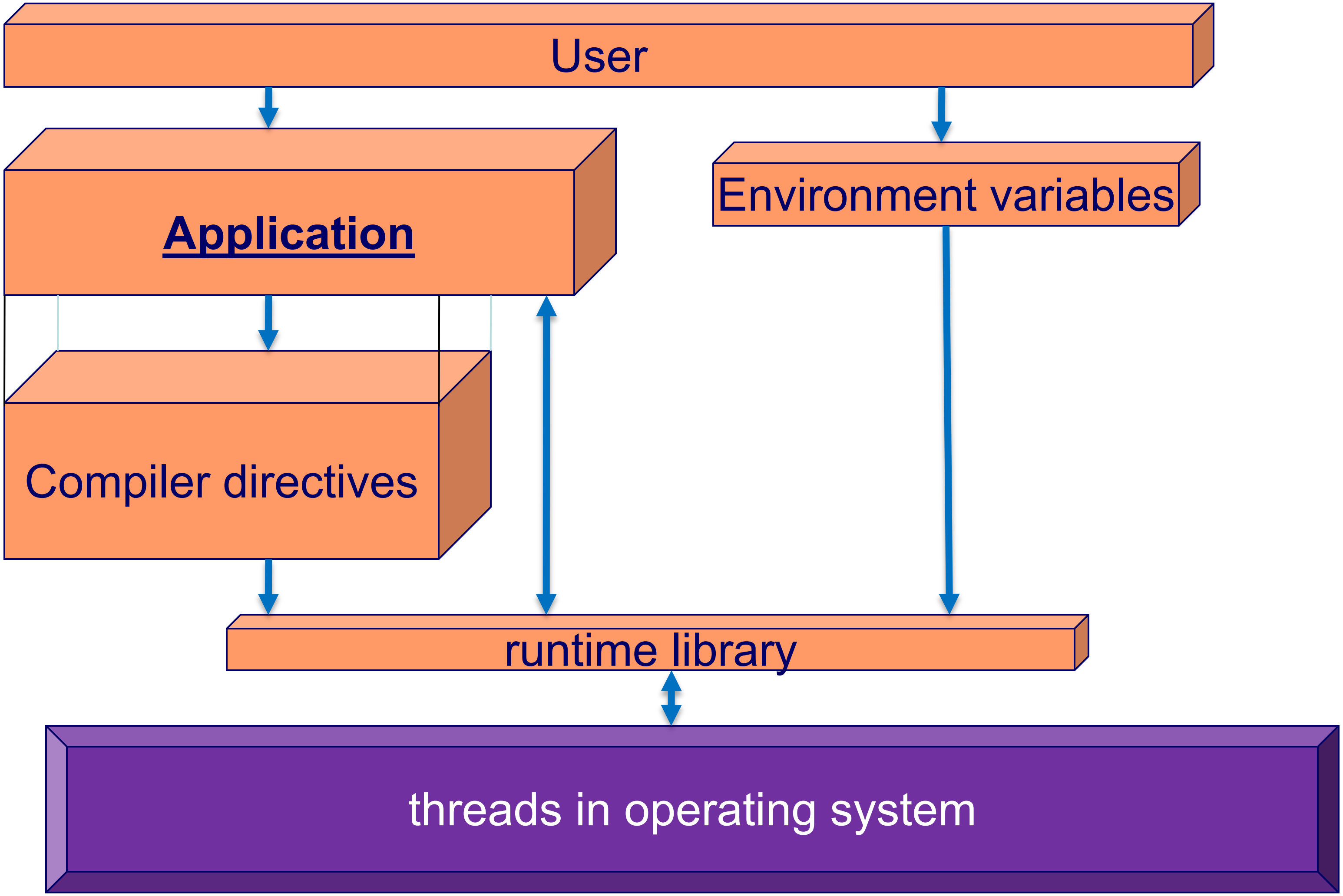
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually you would not do this. (But TS with user threads is less expensive than TS with processes).

- Implementation of threads differs from one OS to another.

Programming with OpenMP on Shared Memory Systems

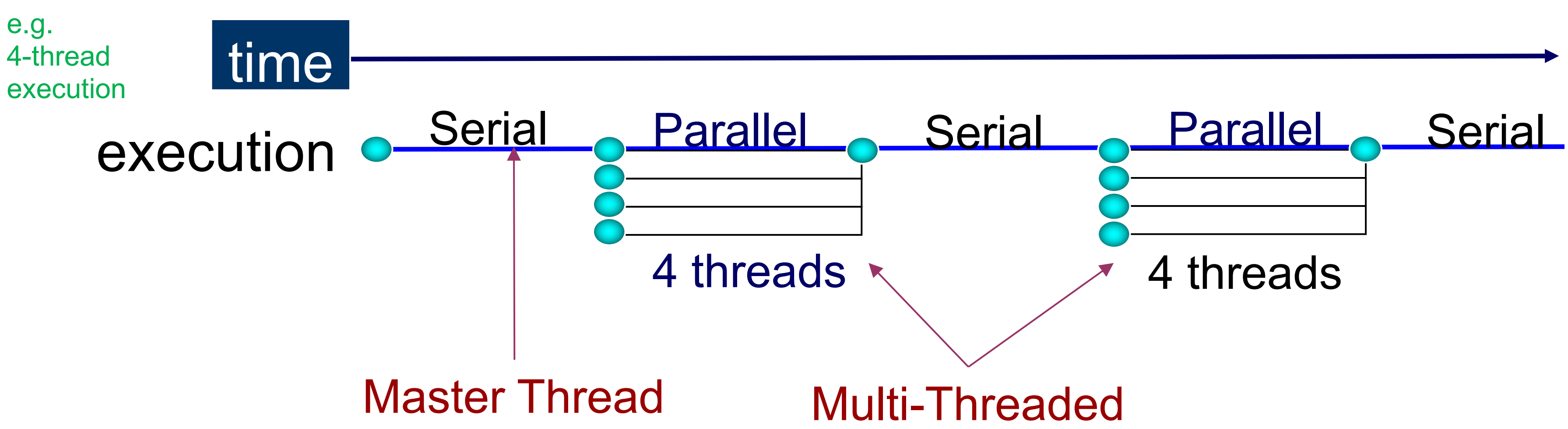


OpenMP Architecture



OpenMP Fork-Join Parallelism

- **Programs begin as a single process**: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread **creates (forks)** a team of parallel **threads** that simultaneously execute tasks **in a parallel region**
- After executing the statements in the parallel region, team threads synchronize and terminate (**join**) but master continues



Intermission

```
a = b + c
```

```
a(i) = b(i) + c(i)
```

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies

What do you have to do?

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies
What do you have to do?

1. Assemble a team
2. Divide up the work
3. Everybody works on their share of the loop
4. Wait for everybody to finish
5. Disassemble team

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies

What do you have to do?

'Plain english'	'OpenMP lingo'
1. Assemble a team	Fork threads
2. Divide up the work	Work sharing
3. Everybody works	*Work* in parallel
4. Wait for everybody	Barrier
5. Disassemble team	Join threads

OpenMP Syntax

- OpenMP Directives: **Sentinel**, **construct** and **clauses**

  #pragma omp ***construct*** **…** **C**

  !$omp ***construct*** **…** **F90**

- **Example for a loop**

  #pragma omp parallel num_threads(4) **C**

  !$omp parallel num_threads(4) **F90**

Loop Example

```fortran
    ...
1. !$omp parallel
1.
2. !$omp do
3. do i=1,n
4.    a(i) = b(i)+c(i)
5. end do
6. !$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    a[i] = b[i]+c[i];
}
}
```

Let's identify the 5 steps:
Fork threads, work-sharing, 'actual work', barrier, join threads

Loop Example

```fortran
    ...
1.  !$omp parallel
1.
2.  !$omp do
3.  do i=1,n
4.     a(i) = b(i)+c(i)
5.  end do
6.  !$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    a[i] = b[i]+c[i];
}
}
```

Let's identify the 5 steps:
Fork threads, work-sharing, 'actual work', barrier, join threads

Forking/joining threads

Work-sharing with implied barrier at the end

2nd Loop Example

```fortran
   ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need?

2nd Loop Example

```fortran
   ...
!$omp parallel

!$omp do
do i=1,n
   a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
   a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
   a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
   a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

1 parallel region

2 work-sharing constructs

How many barriers are there?
How many barriers do we need

2nd Loop Example

```fortran
    ...
!$omp parallel

!$omp do
do i=1,n
   a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
   a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
   a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
   a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need

There is a barrier at the end of the parallel region

There is an implied barrier at the end of the every 'work-sharing' construct'

2nd Loop Example

```fortran
   ...
!$omp parallel

!$omp do
do i=1,n
   a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
   a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```c
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
   a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
   a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need

There is a barrier at the end of the parallel region

There is an implied barrier at the end of the every 'work-sharing' construct'

Total number of barriers: 3

2nd Loop Example (modified)

```
   ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many are necessary to ensure correct results?
How many barriers do we need?

2^{nd} Loop Example (modified)

```
   ...
!$omp parallel

!$omp do
do i=1,n
   a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n
   a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
   a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n;i++){
   a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many are necessary to ensure correct results?
How many barriers do we need?

2 barriers are needed

First: to ensure correct results
Second: to join the threads

End of Intermission